Implementation of MLP for Edge AI

Author: Coby Cockrell
Date: 4/30/2024
Email: cockrellc2@vcu.edu

GitHub: https://github.com/CobyEC/EdgeMLP.git

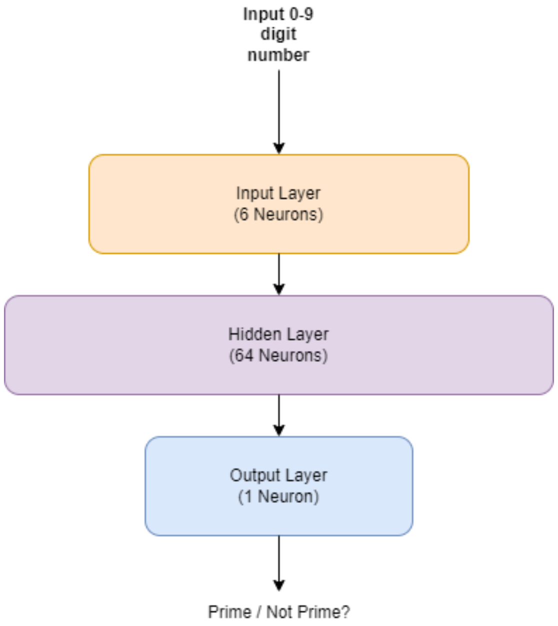
1. Abstract

The purpose of this document is to further investigate and analyze the implementation strategy utilizing Vivado/Vitis HLS for custom Neural Networks / SDeep Neural Networks (SDNN) architectures. As AI/ML takes the world by storm there are more and more edge designs which start incorporating these systems onto their platforms. Many challenges are being seen as of now where the computational demand not only raises issues and errors when interfacing with the edge, but can be extremely costly, as these systems are very rarely power optimized. This is where having the ability to translate your higher level algorithms into a lower, less power hungry, more efficient, faster level becomes ever more crucial. Here I introduce Vivado/Vitis HLS, these embedded design platforms have been slowly expanding their capabilities for years, and have a dual RTL/System Block Designer that integrates in their High-Level-Synthesis tool that translates C/C++ code into RTL/Digital Circuit Logic. These are not easy to understand or use platforms, and often take years to master, however with a bit of luck and certain struggle I aim to demonstrate the power of taking a C++ made Multi Layer Perceptron, and translating most if not all down to the circuit level. Additionally, it should be remarked that not all AI models are capable of being fully 100% realized, as deeper architectures can pose additional challenges to the space of fabric available, this being said it doesn't stop parts of the system being realized into digital circuits and/or equivalent.

2. Objective

The example workflow utilized can be realized and adapted for creating other personalized and fully customized examples. Thus, it is essential to provide clear communication of the design steps, as such lets overview the following problem statement and architecture.

The simple classification chosen for this case is a prime number classification architecture, and as such the architecture will consist of Input layers, Dense layers, and Activation layers. The proposed initial architecture for testing can be seen in the following hierarchy :



3. Initial Design

Since the target HLS solution I've decided to go with is Vitis HLS the initial design language of choice is C++. Utilizing C++ allows Vitis HLS a smooth translation and gives many options for hardware customizations in later steps. For initial design, I have come up with a hierarchy of the C++ and header files that are needed to construct the MLP.

PrimeNum C++ Based System:

Bold = Made and Tested

Prime - Number MLP	headers/	src/	data/	weights/	testbenches/
	mlp.h	**mlp.cpp**	**train.txt**	**weights.txt**	**mlp_Testbench.cpp**
	activation.h	**activation.cpp**	**test.txt**	**bias.txt**	**act_Testbench.cpp**
	layer.h	**layer.cpp**			**layer_Testbench.cpp**
	utils.h	**utils.cpp**			**utils_Testbench.cpp**
		main.cpp			main_Testbench.cpp

For each major component (utils, activation, layer, and mlp) I plan on creating small additional testing scripts to verify each component. This is also a critical step for HLS, as included testbenches are a must. It is also important to keep in mind as this design progresses, the end implementation is an SoC, in which we are able to utilize the PS-PL partitioning to effectively speed-up our system.

After constructing most of the base system, I have thought out and realize now some of the next steps that need to be taken. First and foremost, after successful training and completion of the C++ system constructed, I will need to convert the current training setup into an inference based solution, this allows much less resource draw and is the real way edge AI/ML is implemented. Below I created a table with inference based changes that will need to be made after training.

Inference Based Changes:

Bold = Made and Tested

File(s)	Description
MLP.h / MLP.cpp / MLP_Testbench.cpp	● Remove training-related methods (e.g. backpropagation, weight updates)
	● Add a method to load pre-trained weights and biases
Layers.h / Layers.cpp / Layers_Testbench.cpp	● Remove training-related methods
	● Keep only the forward pass functionality

Main.cpp / Main_Testbench.cpp	● Loads the pre-trained weights and biases
	● Performs inference using the MLP
	● Outputs the predictions